Computers Math. Applic. Vol. 26, No. 2, pp. 61–65, 1993
Printed in Great Britain.

0898-1221/93 $6.00 + 0.00

Analysis of an ADA Based Version of Glassman's General N Point Fast Fourier Transform

B. D. RADUENZ, B. W. SUTER AND E. R. CHRISTENSEN*
Air Force Institute of Technology (AFIT)
Wright-Patterson Air Force Base
Dayton, OH 45433 U.S.A.

(Received and accepted September 1992)

Abstract—An Ada based version of Ferguson's FORTRAN program to compute the general N point fast Fourier transform is provided. Source codes for the two programs are compared and it is demonstrated that the execution time of the Ada program is comparable to that of the corresponding FORTRAN program.

1. INTRODUCTION

Using tensor analysis, Ferguson [1] presented an elegant derivation of Glassman's [2] general N fast Fourier transform (FFT), together with a concise FORTRAN program to implement this FFT. Moreover, most FFT routines that were developed after Ferguson's work [1] were and continue to be based on tensor analysis (see, for example, Van Loan [3]). Nonetheless, most FFT routines available today operate on a vector of length $N = 2^m$, where m is an integer. There are, however, many applications that require a wider choice of N. One of the practical advantages of Glassman's routine is that it can be used in digital signal processing applications for analysis of data of arbitrary length, without the coding complexity of Singleton's case driven routine [4]. The principal disadvantage of Glassman's routine is that it requires an N-vector working space. When programmed in Ada, however, the implementation details of this and other aspects of the code are contained in the body of the FFT package (making them transparent to the user), as shown in the next section.

The high-level software programming language Ada was designed and developed by the Department of Defense in the late 1970's and early 1980's to ensure sound software engineering concepts were employed in the development of military systems. Industry and laboratories were originally somewhat lethargic about commercial use of Ada, but today the market approaches $1.5 billion annually [5]. The growing use of Ada buttresses its well-founded advantages over traditional languages, including increased reliability, readability, testability, and modularity. Despite these acknowledged attributes, many feel Ada's main disadvantage is slow execution time, thereby rendering it not applicable to many engineering applications. This paper will demonstrate that Ada execution time is competitive with the execution time of Ferguson's FORTRAN FFT.

*This research was supported in part by the Air Force Office of Scientific Research under Grant No. AFOSR-616-92-0019.

2. SOURCE CODE COMPARISON

Figure 1 is a listing of Ferguson's FFT subroutines. These subroutines were stored and compiled as separate files. Figure 2 gives the analagous Ada code in a FFT package specification and body, respectively. The source code in the two languages is dissimilar in a few notable areas. First, the FORTRAN program allows implicit assignment from a single index array to a three index array within the *Glassman* subroutine. This assignment is handled by the compiler. In Ada, the same assignment could be coded as a triple nested loop, assigning each element in *Data_Vector* to the appropriate location in a new three-dimensional vector.

```
      subroutine FFT_Sub (N, U, Work, Inverse)
            integer N
            complex U(N), Work(N)
            logical Inverse
            integer A, B, C
            logical Inu
            A    = 1
            B    = N
            C    = 1
            Inu  =. true.
C
10          if (B .GT. 1) goto 30
            if (Inu) return
            do 20 i = 1, N
                  U(i) = Work (i)
20          continue
      return
30          A = C * A
            do 40 C = 2, B
                  if (mod(B,C) .EQ. 0) go to 50
40          continue
50          B = B / C
            if (Inu)          call Glassman(A, B, C, U, Work, Inverse)
            if ( .Not. Inu)   call Glassman(A, B, C, Work, U, Inverse)
            Inu = .Not. Inu
            go to 10
      end
C-----------------------------------------------------------------------
C-----------------------------------------------------------------------
      subroutine Glassman (A, B, C, Uin, Uout, Inverse)
            integer A, B, C
            complex Uin(B,C,A), Uout(B,A,C)
            logical Inverse
C
C           This subroutine is called from FFT_Sub
C
            Complex Delta, Omega, Sum
            Twopi    =    6.28318530717958
            Angle    =    Twopi / float(A*C)
            Delta    =    CMPLX(Cos(Angle), -Sin(Angle))
            if (Inverse)  Delta = CONJG (Delta)
C
            Omega = CMPLX(1.0,0.0)
            do 40                IC = 1, C
                  do 30          IA = 1, A
                        do 20    IB = 1, B
                              Sum = Uin(IB,C,IA)
                              do 10 JCR = 2, C
                                    JC = C + 1 - JCR
                                    Sum = Uin(IB,JC,IA) + Omega * Sum
10                            continue
                              Uout(IB,IA,IC) = Sum
20                      continue
                        Omega = Delta * Omega
30                continue
40          continue
C
      return
      end
```

Figure 1. FORTRAN code for FFT subroutine.

A faster and more elegant assignment can be used if the appropriate element in the single index vector can be accessed based on the values of the three indices. It can be shown that some variable

```
with Complex_Pkg;          use Complex_Pkg;
with Type_Package;         use Type_Package;
with Math;                 use Math;

package FFT_Pack is
    procedure FFT (  FFT_Data          : in out Complex_Vector ;
                     Inverse_Transform : in boolean            );
end FFT_Pack;
```

```
package body FFT_Pack is
    procedure Glassman (  A, B, C           :     in integer;
                          Data_Vector       :     in out Complex_Vector ;
                          Inverse_Transform :     in boolean      ) is

        Temp                  : Complex_Vector(1..A*B*C);
        Counter,JC            : integer := 0;
        Two_Pi                : constant float := 6.28318530717958;
        Del, Omega, Sum       : Complex;
        Angle                 : float;
        C_Plus_1              : integer := C + 1;

    begin
        Angle    := Two_Pi / (float(A*C));
        Del      := Complex_Of((Cos(Angle)), (-(Sin(Angle))));

        if (Inverse_Transform) then
                Del := Conjugate(Del);
        end if;
        Omega := Complex_Of(1.0,0.0);

        for IC in 1..C loop
            for IA in 1..A loop
                for IB in 1..B loop
                    Sum := Data_Vector((((IA - 1)*C + (C-1)) * B) + IB);
                    for JCR in 2..C loop
                        JC := C_Plus_1 - JCR; - - No need to add C + 1 each time through loop
                        Sum := Data_Vector((((IA - 1)*C + (JC - 1))*B) + IB) + (Omega * Sum);
                    end loop; - - JCR
                    Counter := Counter + 1;
                    Temp(Counter) := Sum;
                end loop; - - IB
                Omega := Del * Omega;
            end loop; - - IA
        end loop; - - IC
        Data_Vector := Temp; - - assign output back to Data_Vector
    end Glassman;
```

```
    procedure FFT ( FFT_Data           : in out Complex_Vector;
                    Inverse_Transform  : in boolean            ) is
        A   :   integer := 1;
        B   :   integer := FFT_Data'length;
        C   :   integer := 1;
    begin - - FFT
        while (B > 1) loop                  - - define the integers A, B, and C
            A := C * A;                     - - such that A*B*C = FFT_Data'length
            C := 2;
            while (B mod C) /= 0 loop
                C := C + 1;
            end loop;
            B := B/C;                       - - B = 1 causes exit from while loop
                Glassman (A,B,C, FFT_Data, Inverse_Transform);
            end loop;

            if Inverse_Transform then - - optional 1/N scaling for inverse transform only
                for i in FFT_Data'range loop
                        FFT_Data(i) := FFT_Data(i) / float(FFT_Data'length);
                end loop;
            end if;
    end FFT;
end FFT_Pack;
```

Figure 2. Ada code for FFT package.

Data_Vector_3Index(k, j, i) can be represented as *Data_Vector*$((i-1) * B * C + (j-1) * B + k)$ where A, B and C are the integers passed to *Glassman*. The above expression can be simplified to *Data_Vector*$(((i-1) * C + (j-1)) * B + k)$. The assignments shown in Figure 2 were derived by substituting the appropriate indices into this expression. When writing to the vector *Temp*, which becomes the output, the indices of the three index vector used in Figure 1 match the order of the nested loops, and this allows for an Ada assignment via a simple counter variable.

Second, the user of the Ada version need only pass a complex data vector and boolean inverse operator to *FFT*. Work space is established within the appropriate subprocedure, and the vector length N can be determined using Ada array attributes, thereby eliminating two of the passed parameters. Because work space in Ada is not defined at the top level, there is no need to call *Glassman* using a boolean operator and alternating if statements (passing either U or *Work* as the input vector), as is done in the FORTRAN routine. The only reason to pass two arrays would be to maintain integrity of the input data while passing data out as a separate vector.

Finally, the Ada code contains package calls to *Complex_Package*, *Type_Package*, and *Math* for standard complex number manipulations, global type declarations, and mathematical operations, respectively, as well as a $1/N$ scaling at the end of subprocedure *FFT* for the inverse transform. All timing analysis was performed using the forward transform so no scaling would be invoked in either routine. In terms of floating point operations, the Ada and FORTRAN routines are equivalent.

Table 1. Ada vs. FORTRAN execution time comparison.

	No Output		Write to File	
	Average CPU Time in Seconds			
Points	Ada	FORTRAN	Ada	FORTRAN
500	0.1	0.1	0.1	0.3
1000	0.2	0.3	0.3	0.6
2000	0.4	0.6	0.8	1.3
3000	0.7	0.9	1.2	1.9
4000	0.9	1.2	1.7	2.6
5000	1.2	1.5	2.2	3.3
6000	1.5	1.8	2.7	3.8
7000	1.8	2.2	3.2	4.6
8000	2.0	2.5	3.6	5.2
9000	2.3	2.8	4.1	5.8
10000	2.6	3.2	4.6	6.5
11000	3.2	3.8	5.5	7.4
12000	3.1	3.8	5.6	7.8
13000	4.0	4.7	6.7	9.0
14000	3.9	4.7	6.9	9.3
15000	4.0	4.9	7.2	9.9

3. EXECUTION TIME COMPARISON

Execution times for the two programs were compared using CPU time from the Unix *time* command. Although elapsed CPU times are measured to 1/50 of a second with this facility, the exact execution time for the FFT's is not of great importance and is highly machine-dependent. The desired result was a relative measure of FORTRAN and Ada execution time for digital signal processing algorithms such as the one used here.

Runs were made on a Sun SPARCstation 2 (operating system version SunOS 4.1.2), with very light additional load, at the Air Force Institute of Technology. The software packages used in this comparison were Sun FORTRAN (Enhanced FORTRAN 77) Version 3.1.1 and Verdix Ada Version 6.0.3(d). The results are given in Table 1. Notice that executables were developed which

provided no output or wrote to a file to isolate the effects of I/O in the comparison. As is shown, the Ada execution times are comparable to the FORTRAN execution times.

Although run times were roughly the same, the Ada compilation time was noticeably longer than that for FORTRAN. One reason for this extended compilation time is that Ada was developed as a strongly typed language, and performs numerous compilation checks to minimize run time errors. Ada also performs run time checks such as numeric range checking, which can be disabled using the *-S* command. Because FORTRAN does not perform this level of run time checking, the final Ada executable used in this comparison was compiled with run time checks disabled. A significant speed increase could be realized in the FORTRAN program by compiling with the *-fast* or *-fnonstd* commands. However, this compilation results in floating point outputs which do not conform to IEEE standards. Thus, neither the *-fast* FORTRAN nor the *-O* Ada optimization options were used during compilation. It is important to note that the accuracy of both programs outputs was comparable, since the outputs agreed down to the roundoff of the machine.

4. SUMMARY

We have provided a concise Ada version of Glassman's FFT. Inherent differences in Ada and FORTRAN account for code differences in the two programs. Execution time comparisons indicate that Ada is competitive with FORTRAN and is a viable language for digital signal processing applications.

REFERENCES

1. W.E. Ferguson Jr., A simple derivation of Glassman's general N fast Fourier transform, *Computers Math. Applic.* **8** (6), 401–411 (1982).
2. J.A. Glassman, A generalization of the fast Fourier transform, *IEEE Trans. Comput.* **C-19**, 105–116 (1970).
3. C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, Society for Industrial and Applied Mathematics, Philadelphia, (1992).
4. R.C. Singleton, An algorithm for computing the mixed radix fast Fourier transform, *IEEE Trans. Audio Electroacoust.* **AU-17**, 93–103 (1969).
5. D. Dick, Despite resistence, Ada gains respectability, *Signal Magazine* **45** (11), 92–94 (1991).